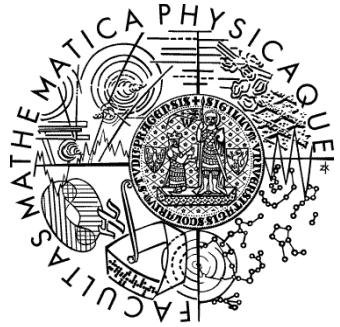
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Peter Szépe

Procházky v grafech a genetické algoritmy

Katedra aplikované matematiky

Vedoucí bakalářské práce: RNDr. Ondřej Pangrác, Ph.D.

Studijní program: Informatika, programování (IP)

2010

Walks in graphs and genetic algorithms

Chtěl bych poděkovat mému vedoucímu RNDr. Ondřeji Pangráci, Ph.D. za jeho čas a rady, Thomasovi Weisovi za jeho knihu Global Optimization Algorithms, Csabamu Tóthovi a Štefanovi Szépemu za pomoc s anglickým jazykem a své rodině a přátelům za podporu.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejěováním.

V Praze dne: 27. května 2010 Peter Szépe

Název práce: Procházky v grafech a genetické algoritmy
Autor: *Peter Sépe*
Katedra (ústav): Katedra aplikované matematiky
Vedoucí bakalárské práce: *RNDr. Ondřej Pangrác, Ph.D.*
e-mail vedoucího: pangrac@kam.mff.cuni.cz

Abstrakt: Řešíme optimalizační úlohy nalezení maximálního tahu mezi dvěma vrcholy v orintovaném grafu s omezením na délku tohoto tahu. Je dán orientovaný graf, startovní a cílový vrchol, délková funkce na hranách a váhová funkce na vrcholech grafu a parametr omezení délky cesty L. Úkolem je najít tah ze startovního do cílového vrcholu celkové délky nejvýše L maximalizující součet vah navštívených vrcholů (každý se započítává pouze jednou).
Tato úloha je NP-těžká a ani aproximační algoritmy nedávají příliš dobré výsledky. Proto je třeba pro praktické aplikace použít heuristické přístupy.

Klícová slova: optimalizace, evolučné algoritmy, genetické algoritmy, grafy, procházky v grafech

Title: Walks in graphs and genetic algorithms
Author: *Peter Szépe*
Department: Department of Applied Mathematics
Supervisor: *RNDr. Ondřej Pangrác, Ph.D.*
Supervisor's e-mail address: pangrac@kam.mff.cuni.cz

Abstract: We solve the problem of finding a maximal walk from the starting vertex to the target vertex with an upper bound of the length of the walk. It is an NP-hard problem where not even the approximation algorithms do guarantee a quick and nice solution. Hence it is important to develop heuristics for practical applications.

Keywords: optimization, evolutionary algorithms, genetic algorithms, graphs, walks in graphs

Contents

Chapter 1

The Problem

We solve the problem of finding a maximal walk from the starting vertex to the target vertex with an upper bound of the length of the walk. It is an NP-hard problem where not even the approximation algorithms do guarantee a quick and nice solution. Hence it is important to develop heuristics for practical applications.

As input we use three parameters: an edge- and vertex-weighted directed graph, a starting vertex, a target vertex and finally a limit for the length of the walk. The goal of the process is to maximize the sum of vertex weights on the walk while every vertex weight can be counted at most once.

By solving this problem, we can plan transports, sightseeing. For example let's take we are tourists in Prague planing a sightseeing, that we would like to finish with a beer in a given pub. We know the distances between the sights and the importance of seeing a particular sight from our guidebook. Our staring point will be the hotel, where we are accommodated, the target is given by the pub and we have 6 hours for this walk. To put the most into our day we have to decide where shall we go first, and on which direction shall we continue. If we have this program, we simply create or load a graph and it plan an optimal/suboptimal walk for us.

Chapter 2

Traditional and Alternative Methods

Beside traditional methods in problem solving we often use nature and biology inspirited techniques.

As a traditional solution for a given problem, we are trying to use the mathematical method:

- We translate formulae, equations, mathematical expressions into computerized running environment. With building up the algorithms upon each other, we are trying to get the solution by using the computer's faultless calculation ability.
- We try to find the solution with the computational velocity of the informatics devices.

At these methods we mostly use deterministic (we can predict the execution time) codes. We have to develop a new solution for every problem. For example we can't use the same formulae to solve a statistical calculation and a static calculation. Moreover, if the basic principle is modified, we have to write a whole new program.

The problems that can be solved with these methods are: simulations, equations, physico-mathematical calculations.

Why do we need alternative methods? We can come across situations, where such problems might occur:

- something is too general or too specific. There is no such mathematical formula that would solve the issue.
- there are too many variables in the "equation" to be solved
- we do not know and/or we can not outline the relations between the variables
- it is necessary to examine too many input values with a linear method while solving the problem
- the input values may change, therefore the system might turn into a non-usable without learning

The alternative methods work on a widely different manner, compared to the traditional methods: The central idea is based on natural science.

The problems that can be solved by this method are: control, management, optimum calculations, and all areas onto which we cannot map an exact solution.

The common features of alternative methods are:

- these may be general
- these may be slow
- we can not predict the execution time, and/or we cannot declare whether it has ended or it will ever end, since we can not determine that the given solution is optimal
- they work using various state spaces
- we influence these state with one or more Back error Propagation algorithms
- the accommodating algorithms are able to learn
- they can get stuck in so-called local minimums, which is not an optimal solution
- these might have different outcomes in the other runs because of random operations
- these are usually using some correction processes
- it is an enormous benefit, that we can create parallel code with threads and child processes. For example, when using genetic algorithms we could have more parallel subsystems with the same specification. The subsystems can run independently from each other with a large population. After a number of iterations we can use crossover between populations or we can select the fittest individuals from each populations to a new population.
- the alternative methods are combinable.

Chapter 3

Evolutionary algorithms

In the following two chapters we will explain the evolutionary algorithms and the genetic algorithms based on the book Weise [11].

”Evolutionary algorithms (EAs) are population-based metaheuristic optimization algorithms that use biology-inspired mechanisms like mutation, crossover, natural selection, and survival of the fittest in order to refine a set of solution candidates iteratively.” Weise [11]

3.1 Some Historical Facts

3.1.1 The basic principles from nature:

Darwin published his book Darwin [2]. The driving force behind the biological evolution is the natural selection and the survival of the fittest.
“His theory can be condensed into ten observations and deductions:

1. The individuals of a species posses great fertility and produce more offspring than can grow into adulthood.
2. Under the absence of external influences (like natural disasters, human beings, etc.), the population size of a species roughly remains constant.
3. Again, if no external influences occur, the food resources are limited but stable over time.
4. Since the individuals compete for these limited resources, a struggle for survival ensues.
5. Especially in sexual reproducing species, no two individuals are equal.
6. Some of the variations between the individuals will affect their fitness and hence, their ability to survive.
7. A good fraction of these variations are inheritable.
8. Individuals less fit are less likely to reproduce, whereas the fittest individuals will survive and produce offspring more probably.
9. Individuals that survive and reproduce will likely pass on their traits to their offspring.
10. A species will slowly change and adapt more and more to a given environment during this process which may finally even result in new species.” Weise [11]

3.1.2 The family of evolutionary algorithms

The idea of using nature inspired selection techniques when optimizing problems has been suggested in the sixties. There were several independent attempts to solve the above mentioned issue.

The German I. Rechenberg has introduced the evolutionary strategies in Rechenberg [7]. It has been used in order to optimize the actual parameters of airplane wings. Hans-Paul Schwefel has improved the method. The evolutionary strategies are being used to independently develop the subdivision of evolutionary algorithms nowadays.

All other techniques are coming from the U.S.A.

L. J. Fogel, A. J. Owens and M. J. Walsh were experimenting with the automatic development of finite-state automatons being used for the solution of simple problems. They have published the results of their experiments in Fogel [3]. They have randomly modified (also known as mutations) the matrix of state changes of the initial automatons. If the new automaton was more suitable, it has been selected. This new technique was named evolutionary programming.

The genetic programming is a more fresh but similar subdivision. The objective of genetic programming is to automatically develop a solution for a specific problem based on the information we have. (mostly in LISP language) The technique was introduced by John R. Koza in Koza [5] and Koza [6].

The genetic algorithms were introduced by John H. Holland in Holland [4]. He summarized his students' research results at University of Michigan. Initially he didn't want to develop an optimization technique, he was just trying to simulate the natural selection and adaption using computers. His results have been criticized in many occasions.

3.2 Introduction to the Evolutionary Algorithms

3.2.1 Used terminologies

We are going to use the following terminologies:

- *Pop* – population
- *Mate* – mating pool, contains individuals who will produce offspring
- *p* – individual
- *g* – genotype
- *p.g* – genotype of the individual *p*
- *x* – phenotype
- *p.x* – phenotype of the individual *p* (solution candidate)
- *gpm* – genotype-phenotype mapping ($x = gpm(g)$)
- *F(p.x)* – vector of objective values of the individual *p*
- *f(p.x)* – objective value of the individual *p*
- *v(p.x)* – fitness value of the individual *p*
- *cmpF* – comparator function, which is used to compare the

individuals
- *Op* – set of reproduction operations

3.2.2 Genotypes and phenotypes

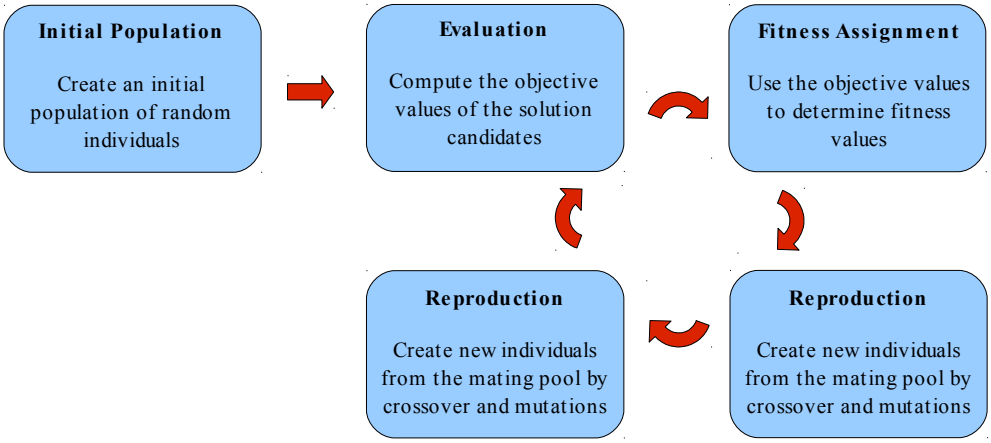
Evolutionary algorithms abstract from Darwin's biological process. The search space $\boldsymbol{G}$ in evolutionary algorithms is a set of all possible DNA strings and its elements $g \in \boldsymbol{G}$ are the genotypes. We often refer to $\boldsymbol{G}$ as the genome and to the elements $g \in \boldsymbol{G}$ as genotypes. All of the creatures are instances of their genotypes formed by embryogenesis. The phenotypes (solution candidates) $x \in \boldsymbol{X}$ in the problem space $\boldsymbol{X}$ are instances of genotypes formed by the genotype-phenotype mapping: $x = gpm(g)$. Their fitness is rated according to objective functions which are subject to optimization and drive the evolution into specific directions.

3.2.3 Simple and multi-objective evolutionary algorithms

Evolutionary algorithms can be divided to single-objective and multi-objective evolutionary algorithms (MOEA). If we try to optimize problem with simple criteria, we use simple-objective evolutionary algorithm. In multi-objective evolutionary algorithms we try to optimize multiple, possible conflicting criteria.

3.2.4 The basic cycle of evolutionary algorithm

1. Initially we create an initial population *Pop* of individuals *p* with a random genome *p.g*.
2. The values of the objective functions $f \in F$ are computed for each solution candidate *p.x* in *Pop*.
3. Using the objective functions a fitness values *v(p.x)* can be assigned to each individual. This fitness assignment process can, for instance, incorporate a prevalence comparator function *cmpF* which uses the objective values to create an order amongst the individuals.
4. A subsequent selection process filters out the solution candidates with bad fitness and allows those with good fitness to enter the mating pool with a higher probability. There is a convention that individuals with smaller fitness value are better, so we try to find the individual with the smallest fitness value.
5. In the reproduction phase, offspring is created by varying or combining the genotypes *p.g* of the selected individuals $p \in Mate$ by applying the search operations $searchOp \in Op$ (which are called reproduction operations in the context of EAs). The better individuals from those offspring are integrated into the next generation of population.
6. If the termination criterion is met, the evolution stops here. Otherwise, the algorithm continues at step 2.



3.2.5 Termination criterion

The termination criterion may be:

- limit for computation time
- total number of iterations
- no improvement in the solution quality could be detected for a specified number of iterations
- if the algorithm has already yielded a sufficiently good solution.

3.2.6 Populations in evolutionary algorithms (generational and elitist evolutionary algorithms)

Interesting is how the population *Pop(t + 1)* of the next iteration is formed as a combination of the current one *Pop(t)* and its offspring. According to this evolutionary algorithms can be divided to two types:

- If the next generation only contain the offspring of the current one, we are talking about generational evolutionary algorithm.
- If at least one copy of the best individual(s) of the current generation is propagated on to the next generation, we are talking about elitist evolutionary algorithm.

3.2.7 Fitness assignment

In multi-objective optimization, each solution candidate *p.x* is characterized by a vector of objective values *F(p.x)*. Many selection algorithms however cannot work with such vectors and need scalar fitness values instead. By assigning a single real number *v(p.x)* (the fitness) to each solution candidate *p.x*, also a total order is defined on them.

A fitness assignment process creates a function $v: \boldsymbol{X} \rightarrow \boldsymbol{R}^{+}$ which relates a scalar fitness value to each solution candidate in the population *Pop*.

There is an convention, that the individuals with smaller fitness value are better solution candidates.

There are more methods for fitness assignment:

- Weighted Sum Fitness Assignment: The most primitive fitness assignment strategy would be assigning a weighted sum of the objective values.
- Pareto Ranking: to each individual, we can assign a value inversely proportional to the number of other individuals it prevails
- Tournament Fitness Assignment: the fitness of each individual is computed by letting it compete q times against r other individuals (with r = 1 as default) and counting the number of competitions it loses
- Variety Preserving Ranking

3.2.8 Selection

In evolutionary algorithms, the selection operation *Mate = select(Pop, v, ms)* chooses *ms* individuals according to their fitness values *v* from the population *Pop* and places them into the mating pool *Mate*.

Selection may behave in a deterministic or in a randomized manner, depending on the algorithm chosen. Many selection algorithms only work with scalar fitness and thus need to rely on a fitness assignment process in multi-objective optimization.

The selection algorithms are:

- Truncation Selection: Truncation selection, also called deterministic selection or threshold selection, returns the $k < ms$ best elements from the list *Pop*. These elements are copied as often as needed until the mating pool size ms reached.
- Fitness Proportionate Selection: Fitness proportionate selection has already been applied in the original genetic algorithms as introduced by Holland [4] and therefore is one of the oldest selection schemes. In fitness proportionate selection, the probability $P(p_1)$ of an individual $p_1 \in Pop$ to enter the mating pool is proportional to its fitness $v(p.x)$ compared to the sum of the fitness of all individuals.
- Tournament Selection: Tournament selection, proposed by Wetzel [12] and studied by Brindle [1], is one of the most popular and effective selection schemes. Its features are well-known and have been analyzed by a variety of researchers. In tournament selection, k elements are picked from the population *Pop* and compared with each other in a tournament. The winner of this competition will then enter mating pool *Mate*. Although being a simple selection strategy, it is very powerful and therefore used in many practical applications.
- Ordered Selection: Here, the probability of an individual to be selected is proportional to (a power of) its position (rank) in the sorted list of all individuals in the population.
- Ranking Selection: the probability of an individual to be selected is proportional to its position (rank) in the sorted list of all individuals in the population.
- VEGA Selection: The Vector Evaluated Genetic Algorithm by Schaffer [8, 9] applies a special selection algorithm which does not incorporate any preceding fitness assignment process but works on the objective

values directly. For each of the objective functions $f_i \in F$, it selects a subset of the mating pool *Mate* of the size $ms/|F|$.

3.2.9 Reproduction

The reproduction operations will subsequently be applied on the mating pool. Optimization algorithms use the information gathered up to step t for creating the solution candidates to be evaluated in step $t + 1$.

There are four basic reproduction operations:

- Creation has no direct natural paragon; it simple creates a new genotype without any ancestors or heritage.
- Duplication resembles the cell division, resulting in two individuals similar to one parent.
- *Mutation* in evolutionary algorithms corresponds to small, random variations in the genotype of an individual, exactly like its natural counterpart.
- Recombination combines two (or more) parental genotypes to a new genotype including traits from both elders.

Chapter 4

Genetic Algorithms

Genetic algorithms (GAs) are a subclass of evolutionary algorithms where the elements of the search space are binary strings or arrays of other elementary types.

Some example areas of application of genetic algorithms are: scheduling, chemistry, chemical engineering, medicine, data mining, data analysis, geometry, physics, economics and finance, networking and communication, electrical engineering and circuit design.

4.1 String chromosome

A string chromosome can either be a fixed-length tuple or a variable-length list.

In the first case, the loci i of the genes g_i are constant and, hence, the tuples may contain elements of different types $\boldsymbol{G}_i$.

$\boldsymbol{G} = \{g = (g[1], g[2], .., g[n]) \mid g[i] \in \boldsymbol{G}_i, i = 1..n\}$

This is not given in variable-length string genomes. Here, the positions of the genes may shift when the reproduction operations are applied. Thus, all elements of such genotypes must have the same type $\boldsymbol{G}_T$.

$\boldsymbol{G} = \{g = (g[1], g[2], .., g[\mathit{length}(g)]) \mid g[i] \in \boldsymbol{G}_T, i = 1..\mathit{length}(g)\}$

4.2 Behavior of search operations

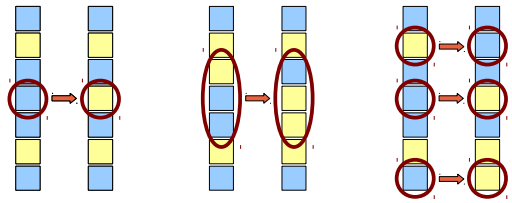
Genetic algorithm provide search operations which closely copy sexual and asexual reproduction schemes from nature. In such “sexual” search operations, the genotypes of the two parents genotypes will recombine. In asexual reproduction, mutations are the only changes that occur. It is very common to apply both principles in conjunction, i. e., to first recombine two elements from the search space and subsequently, make them subject to mutation.

4.3 Reproduction on fixed-length string chromosomes

We use reproduction operators to create new solution candidates to the next generation. Those operations are inspired by the biological procreation mechanisms of nature. There are four basic operations:

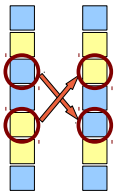
- Creation: Creation of fixed-length string individuals means simple to create a new tuple of the structure defined by the genome and initialize it with random values.

- Mutation: this can be achieved by randomly modifying the value of a gene, group of genes or modifying values of more genes.



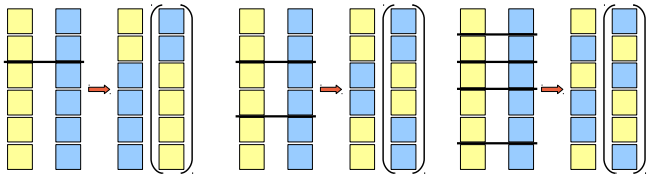
Single-gene mutation Multi-gene mutation Multi-gene mutation

- Permutation: is an alternative mutation method where the alleles of two genes are exchanged. This makes only sense if all genes have similar data types.



Permutation

- Crossover: is performed by swapping parts of two genotypes. This comes closest to the natural paragon.

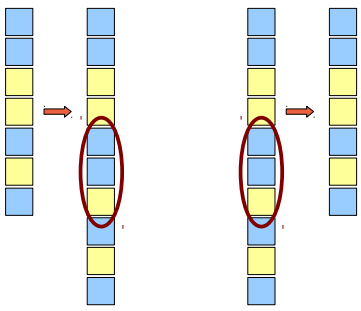


Crossover

4.4 Reproduction on variable-length string chromosomes

Variable-length genomes were introduced by Smith [11]. He used it for a program which plays poker. At variable-length chromosomes strings need to be constructed of elements of the same type, so there is no relation between locus and type. We have the following reproduction operations:

- Creation: variable-length chromosome is created by random drawing a length > 0 then create a randomly filled list of that length.
- Mutation: the set of mutations introduced in fixed-length chromosomes can be extended by insertion of couple randomly chosen genes and by delete a randomly length sequence from the chromosome.



Mutation using insertion Mutation using delet

- Crossover: the same crossover operations are available as for fixed-length strings except that the strings are no longer necessarily split at the same loci.

Chapter 5

The Solution

There exists a lot of possible solutions for this problem. Some solutions are slower, but we will certainly get the maximal walk, others are faster for finding a sub optimum, but it is uncertain whether we will get the maximal walk.

5.1 The simplest and maybe the slowest solution

The simplest and most obvious solution is backtracking. This is a very slow method, the algorithm tries all possible walks and returns with the maximal one.

To implement it we need a bit modified adjacency matrix of the graph, where the non-diagonal entry a_{ij} means the length of the edge from vertex i to vertex j represented by a natural number. In case there is no edge from vertex i to vertex j, a^{ij} should be NULL. Above this we need a list, which will represent the walk.

At the beginning we add the starting vertex to the list. We call a recursive function, which ends, if the length of the walk reaches the given length limit . Let the last vertex from the list(walk) be vertex i. In every recursive function we test if there exists an edge from vertex i to other vertices from the graph. We can get this information from the i^{th} row of the adjacency matrix. If the j^{th} entry is not NULL, we add the vertex j to the end of the list and call the recursive function. If the j^{th} entry is NULL, we try the $j+1^{th}$ and so we continue. When we add the target vertex to the list, we test if this walk is better than the best result achieved so far.

This solution could be very slow because of the amount of decisions it has to make. Moreover, it could be possible, that we add vertices to the list, which are reached from the starting vertex, however the target vertex is not reachable from those ones.

The algorithm can be improved by using an array, which contains on the i^{th} position an array of veritec, which are reachable form vertex i and the target vertex is the one reached from those vertices. We can simply fill this array by using the result of the Floyd-Warshall algorithm.

5.2 Fixed-length genetic algorithm

The most difficult part of solving this problem by fixed-length GA is the representation of the individuals(walks) by genotype.

From this point of view the Travel Sales Problem is much easier, the count of verices on the path is fixed and it equals the count of vertices in the graph,

therefore we can easily declare the length of the genotype as the count of vertices in the graph. Actually the genotype will be the path itself, so we don't have to use difficult genotype-phenotype mapping.

In our case the count of vertices on the walk can and should be variable, so it could not help us to declare the length of genotype. We have to declare a suitable length for the genotype, which is either not too short neither too long. For example the length could be the number of vertices in the graph or for example the number of vertices / 2.

The first entry of the genotype must be the starting vertex while the last entry has to be the target vertex. We can randomly fill other entries. For genotype-phenotype mapping we use the shortest paths between vertices in the genotype. After that we have to eliminate the sequences of same vertices from the walk. The crossover could be implemented by randomly splitting the genotype and swapping parts of two genotypes. The mutation can be achieved by randomly modifying the value of the gene at random position or by randomly modifying the genes with given probability.

5.3 Variable-length genetic algorithm

This is the most obvious solution, here we don't need complicated operations for genotype-phenotype mapping. We can represent the genotype by a sequence of vertices on the walk.

The only disadvantage of this method is that we need quite complex reproduction operators. For example mutation can be implemented as follows:

We are going to modify a randomly chosen gene (vertex at a given position in the genotype). Let this gene be on the i^{th} position in the genotype. We have to modify this gene and connect it with the rest of genotype using shortest paths. If we randomly modify the gene, it can happen, that there is no path from the $i\text{-}1^{th}$ gene to the new vertex or from the new vertex to the $i+1^{th}$ gene. To prevent this situation, we have to chose the new vertex from a list of verices that can be connected with the rest of the genotype.

We can use similar principles to implement crossover.

In the following sections I will go into details about solving the problem with the variable-length genetic algorithm.

Chapter 6

Graph Representation

There exists several data structures for graph representation. For different algorithms the required data structures also differ.

6.1 Adjacency matrix

The most common data structure is adjacency matrix. In our case the adjacency matrix is not the best choice for the optimization, but we will use an adjacency matrix to create more adequate data structures.

6.2 Results of floyd-warshall algorithm

Running every time a Dijkstra algorithm to find shortest paths between the vertices will be expensive. So we need a matrix which contains the result of the Floyd-Warshall algorithm and a second one where the entry a_{ij} means the penultimate vertex on the shortest path from i to j.

6.3 Matrix of the shortest paths

In our algorithm we will need the shortest paths between selected vertices, so the fastest implementation requires a dedicated matrix, where the nondiagonal entry a_{ij} contains the shortest path from vertex i to vertex j. The diagonal entry a_{ii} contains a single element list with vertex i.

We can create this matrix using the result of the Floyd-Warshall algorithm ran on penultimate vertices. Let *k* be the penultimate vertex on the shortest path from vertex *i* to vertex *j*. Then the shortest path will end with this sequence of vertices: *k*, *j*. So to get the shortest path, we have to continue searching with penultimate vertex on the shortest path from vertex *i* to vertex *k*. This should be repeated until we reach vertex i.

6.4 Lists based on reachability

To implement faster search operators we need lists of vertices that are based on reachability.

We need a list that contains the vertex *v* only if *v* is reachable from the starting vertex and the target vertex is also reachable from vertex *v*. Let this list be called *ListOfComponents*.

We need an array, where the entry a_i means a list containing the vertex *v* only if *v* is reachable from vertex *i* and *v* is member of *ListOfComponents*. Let this array be called *ArrayOfReachableVertices*.

We also need an array, where the entry a_i means a list, which contains the vertex *v* only if vertex *i* is reachable from *v* and *v* is member of

ListOfComponents. Let this array be called *ArrayOfAntecedentVertices*.

The last data structure we need is a matrix, which is a combination of all the above mentioned arrays. The entry $a_{i,j}$ means a list containing the vertex *v* only if *v* is reachable from vertex *i*, while vertex *j* is reachable from *v* and *v* is member of *ListOfComponents*. Let this matrix be called *MatrixOfMiddleVerices*.

Chapter 7

The individual

The subjects of the following chapters are the individuals. Here will be shown the method of representing genotypes, what is the phenotype, how to implement genotype-phenotype mapping, how to calculate the fitness value of the individual, how to generate initial individuals and of course there will also be some words about, how to implement search operators such as crossover and mutation. After this chapter we will know almost everything to implement the genetic algorithm.

7.1 The Genotype

The genotype is a sequence of vertices representing the walk. The vertices in the genotype are ordered by the time we visit them on the walk. The first and last genes are locked, so those we can't modify. The first gene is the starting vertex, the last gene is the target vertex.

The only criteria of other genes is that there must be an edge from genotype[i] to genotype[i+1] in the graph.

7.2 The Phenotype

In our case the phenotype is actually the walk with its length and sum of vertex weights.

7.2.1 Genotype-phenotype mapping

The genotype-phenotype mapping is very simple, the only thing we need to calculate is the length of the walk and the sum of vertex weights. The walk itself is given by the genotype.

7.2.2 Terms of length of the walk

We will refer to the maximum allowed walk length as *lengthLimit* .

We are going to use two constant factors from the set of the real numbers :

1. *min :* this factor is used at generating the initial population. Length of the randomly generated walk must exceed $min * lengthLimit$.
2. *max* : this factor is used at generating the initial population and also at fitness assignment. Length of the randomly generated walk can't

exceed $max * lengthLimit$. We are trying to eliminate those individuals from populations, which have length of the walk greater then $lengthLimit * max$ using fitness value set to infinity.

The optimal value of those factors will be chosen in section 9.2.

7.3 Fitness Assignment

Despite the fact that we only have to maximize the sum of vertex weights, we have to use a multi-objective genetic algorithm. This is because we have an upper bound for the length of the walk (*lengthLimit*), so we are going to use two objective values. One is calculated from the length of the walk (f_l), the other from the sum of vertex weights (f_s).

7.3.1 The scalar fitness value

The following formula will be used to calculate the fitness value of individual p ($v(p.x)$):

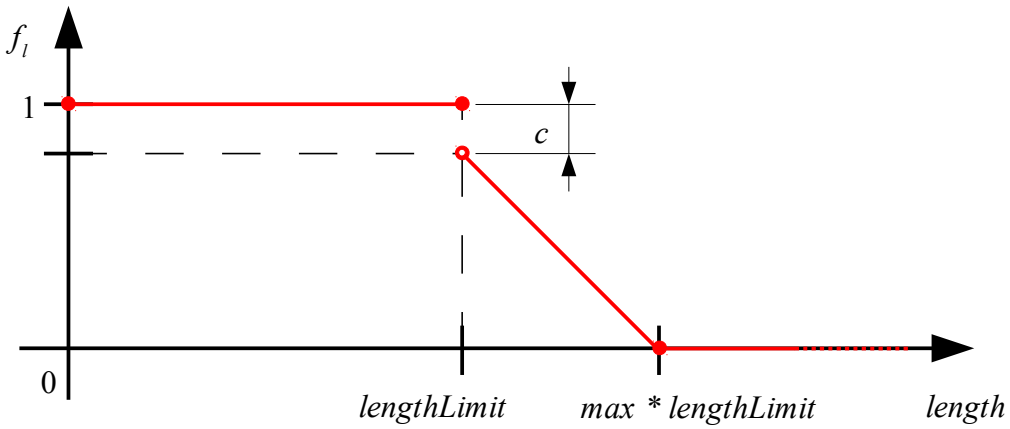
$$v(p.x) = \frac{1}{f_s(p.x) * f_l(p.x)}$$

By definition of fitness value a better individual has lower fitness, so contrary to the conventions f_s and f_l must to be higher at better individuals.

7.3.2 The objective values

We have the following objective values:

1. The objective value calculated from the sum of vertex weights: $f_s(p.x) = weightSum(p.x)$, where $weightSum(p.x)$ means the sum of vertex weights of individual p.
2. the objective value calculated from the length of the walk: calculation of this value is more difficult. We can divided this problem into two parts based on the length of the walk ($length(p.x)$):
   - if $length(p.x)$ is less or equal to $lengthLimit$, we need a constant value, let this value be $f_l(p.x) = 1$.
   - If $length(p.x) > lengthLimit * max$, to satisfy the condition of fitness value defined above, let $lengthFitness = 0$.
   - else we need a suitable formula to calculate the value of $f_l(p.x)$. It would not be beneficial to set $f_l(p.x)$ close to 1, if $length(p.x) = lengthLimit + \varepsilon$, where $\varepsilon > 0$, so we can set $f_l(p.x) = 1 - c$, if $legth(p.x) = lengthLimit_+$, where c is a suitable constant. The value of f_l has to constantly decrease and it has to reach 0 if $length(p.x) = max * lengthLimit$. We will use a linear function for this section.



$$f_l(p.x)=\frac{max*lengthLimit-length(p.x)}{max*lengthLimit-lengthLimit}*(1-c)$$

We get the next formula after simplification:

$$f_l(p.x)=\frac{1-c}{max-1}*\left(max-\frac{length(p.x)}{lengthLimit}\right)$$

Let $q=\frac{length(p.x)}{lengtLimit}-1$, then

$$f_l(p.x)=\frac{1-c}{max-1}*\left(max-\frac{(1+q)*lengthLimit}{lengthLimit}\right)=(1-c)*\left(1-\frac{q}{max-1}\right)$$

Using the above formula in the followings we calculate how „good" f_s has to be of an individual with $length>lengthLimit$ to „beat" an individual with $length\le lengthLimit$. Let these two individuals be called p_1 and p_2 .
Then we have next equation and formulas:

1. $v(p_1.x)>v(p_2.x)$
2. $v(p.x)=\frac{1}{f_s(p.x)*f_l(p.x)}$
3. $f_l(p_1.x)=(1-c)*\left(1-\frac{q}{max-1}\right)$
   where $q=\frac{length(p_1.x)}{lengtLimit}-1$
4. $f_l(p_2.x)=1$

After substitutions and simplifications the following relation is obtained:

$$\frac{f_s(p_1.x)}{f_s(p_2.x)}>\frac{1}{1-c}*\frac{max-1}{max-1-q}\quad ,$$

if $length(p_1.x) < max * lengthLimit$, otherwise this proportion is ∞.

Let's see some proportions:

$q=$	0.00	0.05	0.10	0.20	0.50
$max=1.2,\ c=0.1$	111%	148%	222%	∞	∞
$max=1.2,\ c=0.2$	125%	167%	250%	∞	∞
$max=1.5,\ c=0.1$	111%	123%	139%	185%	∞
$max=1.5,\ c=0.2$	125%	139%	156%	208%	∞
$max=1.5,\ c=0.5$	200%	222%	250%	333%	∞
$max=2.0,\ c=0.2$	125%	132%	139%	156%	250%

$max=1.5$ and $c=0.2$ seems good, but the optimal value of max and c will be chosen in section 9.1.

There is an important task, which has little in common with fitness assignment, but we use f_s in this section of the algorithm. If the f_s is greater than the maximal sum of vertex weights achieved so far, we have to mark f_s as the maximal sum and the genotype as the walk belonging to this sum.

7.4 Generating the Individuals

7.4.1 Randomly generated individuals

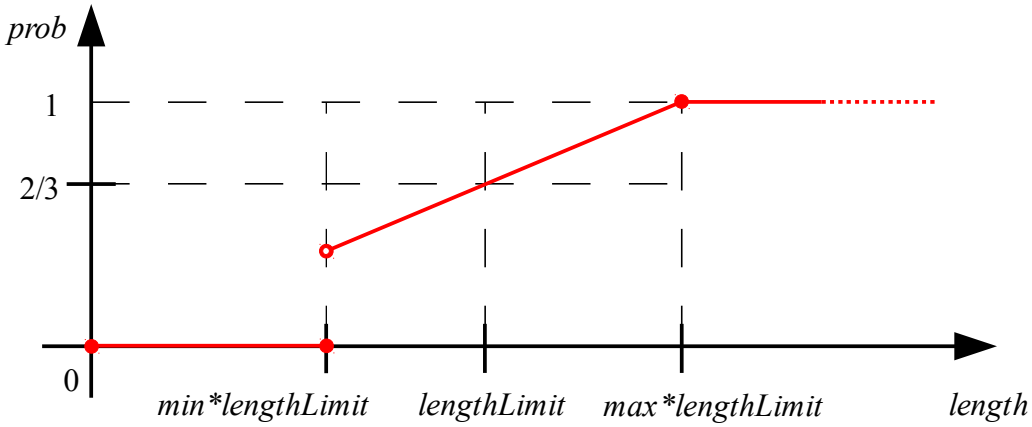
The values of first and last genes are locked in the genotype. There is a little problem with the last gene, it is known that it must be target vertex, but we don't know the loci of this gene. In our case generating the individual is not as easy, as it would be using fixed-length genotypes because we don't know the length of the genotype. So our object is to generate the other genes and if we are ready with them, we have to append the last gene. Let the values of new genes be generated in a loop, so we should somehow declare the length of the randomly generated genotype. If this length is exceeded, we have to leave the cycle. It can be done deterministically or with some probability ($prob$). This probability should be somehow depended on the length of the walk, which is represented by this genotype.

We will use the following method to implement the upper mentioned loop:

1. initially $prob=0$
2. before every iteration of the cycle we randomly generated a real number end from interval 0 to 1. We leave the cycle if $end \le prob$, otherwise we stay in the loop.
3. after generating the value of the new gene, we are calculating the length of the walk ($length(p.x)$), including the last gene, which is the

target vertex.

4. We are calculating the probability ($prob$) of leaving the loop from $length(p.x)$. According to definition of min and max if $length \leq min * lengthLimit$ than $prob$ has to be 0. If $max * lengthLimit < length(p.x)$, it has to be 1. Otherwise a suitable formula is needed to calculate the value of $prob$. We are going to use the following simple function to compute the $prob$:



So if $min * lengthLimit < length(p.x) \leq max * lengthLimit$, then

$$prob = 1 - \frac{1/3}{max * lengthLimit - lengthLimit} * (max * lengthLimit - length(p.x))$$

After simplifications we get this formula:

$$prob = \frac{2}{3} + \frac{length(p.x) - lengthLimit}{3 * lengthLimit * (max - 1)}$$

It is useless to use a complicated function to calculate the probability of leaving the cycle. Having a good initial population isn't as important as the use of good fitness assignment and search operations. We can get a fit population after some iterations even from a very bad initial population. The only have to guarantee, that the individuals will mostly have $length(p.x) \leq lengthLimit$.

5. go to step 2.

After leaving the cycle it should be checked that the length of the walk didn't exceed $max * lengthLimit$. If it did, we have to consider it as an invalid individual.

The only remained task is to calculate the fitness value.

7.4.2 Generating individuals from a given genotype

There are several occasions when we need to generate an individual, whose genotype is given. We have to check that the length of the walk don't exceed $max * lengthLimit$ and calculate the fitness value.

7.5 Crossover

At crossover we have to recombine the genes in genotypes of two individuals. Let this individuals be p_1 and p_2. We have to solve two main problems and one additional:

- How to find the loci, where the genotypes are split?
- How to join the parts of genotypes?
- We have to decide whether we want to produce one or two offspring using crossover.

7.5.1 Finding the loci

At first we have to chose the approximate position of splitting. That means, that we are randomly choosing a percent q, the approximate positions will be $pos_1 = trunc(q * length(p_1.g))$ and $pos_2 = trunc(q * length(p_2.g))$ where $trunc(x)$ means the truncated value of x.

7.5.2 Connecting the parts

We are going to connect the split genotypes using shortest paths between the parts of genotypes. If we split the genotypes at positions pos_1 and pos_2, then we swap the split genotypes using shortest path between the parts of genotypes the walk can easily be too long, because of the fact, that the length of the walk of individuals are mostly near to $lengthLimit$. So we have to modify those positions somehow. We have to declare the number of genes, that we will replace with the shortest path. We will refer to this value as rl (replace length). We will split the genotypes at positions $pos_1 - trunc(rl/2)$ and $pos_2 - (rl - trunc(rl/2))$. We will declare the value of rl in section ???

This seems difficult, but using this method the offspring will have legal length in most cases and thus we increase the speed of the algorithm.

7.5.3 Number of offspring

The number of produced offspring is just a simple parameter, which doesn't modify the whole algorithm, only the speed of algorithm is depended on it. This parameter will be tested in section 9.6.

7.6 Mutation

As the parameters of mutation we have the maximal number of performed mutations, and the number of performed mutations is randomly generated from interval 1 to a maximal number. We will test the chose the maximal number of mutations in section 9.7.

At mutations we have similar problems as at the crossover. We have to chose a suitable number of replaced genes, which must be about the same length as the replacement, where the replacement contains from the new vertex

joined by shortest paths with the primal parts of the genotype. The number of replaced genes is $rg = rl + 1 + rl$. We must to insert a new vertex and connect it with two other vertices. After the connection we have to generate the position where to start eliminating genes from the genotype. This value is a random number from 2 to $(length(p.g) - 1) - rg$ (the first and the last gene of the genotype are locked). After we have eliminated the genes we have to randomly generate the new vertex from the correct position of *MatrixOfMiddleVerices* and connect it with the rest of genotype.

7.7 Number of Replaced Genes

Maybe this is the most important part of the whole optimization algorithm. If we replace the correct number of genes at crossover and mutation processes with new ones, we can greatly increase the speed of the optimization. But after trying lots of formulas and parameters, we have to recognize that there is no formula that will be the best for all problems. There are some values of the graph which influence the correct value of the rl, but there is no general formula or it is very complex. We can easily develop a formula which is suitable for some selected graphs with upper bounds of the walk but then we can also create a graph for which this formula will not be suitable. So our goal is to develop an optimal formula to calculate the value of rl with which our algorithm will run fast in most cases.

7.7.1 Values of the Graph

The structure of the graph has the most influence for the value of rl. We have to calculate the following values:

- The mean of the number of vertices in the shortest paths between all pairs of different verices. We will refer for this value as $mean$. This value is very important because we use shortest paths to connect the parts of genotypes
- The deviations (not the standard deviation) of the number of vertices in the shortest paths from $mean$. At first we divide the set of shortest path to two subsets, first subset ($sSet$) contains those shortest paths which are smaller then $mean$, second subset ($gSet$) contains the rest of the shortest paths. We are going to calculate the generalized mean with exponent 2 of this sets of shortest paths. We will refer for this values as $sMean$ and $gMean$. Also we have to calculate the proportion $pr = |sSet| / (|sSet| + |gSet|)$.
- The average number of edges leading from vertices. We will refer for this value as $avNuEd$

7.7.2 The Formula

We are going to use the following formula to compute the number of genes to replace at crossover and mutation processes:

1. At first we have to decide whether rl will be smaller or greater than

mean value. We have to randomly generate a real number $rand$ from interval 0 to 1. If this number is smaller then pr it means rl will be smaller then $mean$, otherwise rl will be the greater.

2. If $rand < pr$ then

   $rl = mean - sMean * \frac{random(50,150)}{100} * factor$,

   else

   $rl = mean + gMean * \frac{random(50,150)}{100} * factor$.

   Where $factor = \frac{rlFactor * avNuEd^2 * length(p.g)}{1000}$ at mutations and $factor = \frac{rlFactor * avNuEd^2 * \frac{length(p_1.g) + length(p_2.g)}{2}}{1000}$ at crossovers. $rlFactor$ is a parameter which will be tested in ???

   This formula seems very complex but a little change in the value of $factor$ can lead to large differences in the speed of the optimization. So we can imagine how the perfect formula would look like if it even exists.

Chapter 8

The Evolution

Now we know almost everything to build a genetic algorithm. We just have to put together the above discussed methods using selection, reproduction and reinsertion then put it into a loop. So the subjects of this chapter will be selection, reinsertion and the loop of evolution.

We have to define the size of population, and the mating pool. We are going to show the relation between these values and the speed of the optimization in section 9.4.

8.1 Selection

We are going to use tournament selection. In tournament selection, k elements are picked from the population and compared with each other in a tournament. The winner of this competition will then enter mating pool. Because of that, more copies of the same solution candidate can contribute to the mating pool. Better individuals will have more copies in the mating pool and the worst individual probably will not contribute to the mating pool. The number of above mentioned copies depends on the size of k. The relation between k and the speed of the optimization is showed in section 9.5.

8.2 Reproduction

After we create the mating pool, for each individual from the mating pool we have to select a random pair from mating pool as second parent. For each pair of parents we have to create an offspring using crossover. The new individuals must be mutated.

8.3 Reinsertion

The best individuals created by the reproduction operators are propagated to to the next generation. The number of those individuals is the minimum of the size of the mating pool and the number of offspring created by reproduction. The remaining entries of the new population are filled with the

best individuals from the previous population, so if the size of the mating pool is less then the size of population, we use elitist evolutionary algorithm, otherwise it is a generational evolutionary algorithm.

8.4 The Loop

At first we have to generate the initial population what is easy. We need a container to hold the population, let this container be called *Pop*. We have to fill the *Pop* with random generated individuals.

The loop of evolution: The core of the loop contains selection, reproduction and reinsertion. A new generation of solution candidates are created in every iteration, so we get closer and closer to the optimal solution.

If the termination criterion fulfills we will stop the cycle and also the optimization. So we have to chose a suitable termination criterion, which can be:

- A given number of iterations.
- A given number of iterations counted from the last new solution candidate.
- The sum of vertex weights reaches or exceeds a given limit.
- The User stops the optimization.

Chapter 9

Testing of the Parameters

In this chapter we will find the optimal settings for the above defined parameters: *min*, *max*, *c*, *rlFactor*, size of the population (*ps*), size of the mating pool (*ms*), the number of compared individuals at tournament selection (*k*), the maximum number of performed mutations and the number of offspring created ad crossover.

We have to declare how to compare the speed of the optimization. At first we have to calculate the maximal sum of vertex weights of the tested graph with the given upper bound for the length of the walk. Then we have to run the optimization process several times in a row. If the sum of vertex weights achieves the calculated maximal sum then the optimization terminates. At this point we make a note of the time the optimization required. From the elapsed times we are generating statistical data and charts. We draw charts which contain the average state of the optimization with respect of the elapsed time and the inverse of the distribution function. We are going to minimize the time which is assigned to 95% in the inverse distribution function. At some testes we are going to mention some additional information for example the average time, the average time when we achieve 99% of the maximal sum of vertex wights or the above mentioned charts.

For testing we are going to use 5 different graphs which are displayed in appendices in 11.1.

The tests were ran on a PC with Intel Core 2 Duo CPU P8400 (2.26 GHz), 2 GB RAM (1066 MHz DDR3), Microsoft Windows 7 Professional 64 bit operating system and the code was compiled in Microsoft Visual Studio 2008.

9.1 Testing of Fitness Assignment Parameters

At this section we are going to find the optimal values for parameters *max* and *c*. Also there is no such parameter that will be serve as best for all graphs so we have to test those parameters on more graphs and select the one which seems the most reasonable. We will compare the time which is assigned to 95% in the inverse of the distribution function.

Results for graph 1:

c \ *max*	2.5	2.7	2.9	3.1
0.10	337	323	302	289
0.05	283	292	261	304
0.00	334	284	194	270

Results for graph 2:

c \ *max*	2.4	2.5	2.6	2.7
0.15	N/A	5302	N/A	N/A
0.05	3783	3424	3359	3871
0.00	3266	2864	1402	1314

Results for graph 3:

c \ *max*	2.4	2.5	2.6	2.7
0.40	771	N/A	N/A	N/A
0.30	964	819	786	N/A
0.00	1501	N/A	1398	2171

c \ *max*	1.1	1.2	1.25	1.3
0.15	858	735	N/A	N/A
0.10	N/A	785	666	N/A
0.00	824	812	N/A	879

Results for graph 4:

c \ *max*	2.2	2.3	2.5	2.6
0.15	N/A	N/A	969	655
0.05	N/A	691	354	379
0.00	197	N/A	225	388

Results for graph 5:

c \ *max*	2.5	2.7	2.9	3.1
0.20	972	636	368	196
0.15	600	244	175	108
0.10	325	221	98	88
0.00	58	64	116	168

In our case the fields with N/A are not interesting.
After evaluating those data we can choose the parameters. The most reasonable value of *c* is 0 and for *max* is 2.6. The only problem is with graph 3 but the difference from the best value is just polynomial with small multiplier.

9.2 Testing of the Value of *min*

min	Graph 1	Graph 2	Graph3	Graph 4	Graph 5
0.0	319	1768	1655	384	65
0.1	298	1479	1822	355	68
0.2	299	1527	1727	403	59
0.3	253	1527	1902	371	60
0.4	275	1389	1926	416	62
0.5	264	1682	1895	380	64
0.6	299	1769	2012	356	57
0.7	323	1638	1889	407	67
0.8	340	1511	1993	411	78
0.9	315	1375	1741	421	60
1.0	337	1857	1996	377	59

There are no big differences between the results and we can find no tendencies what means we can choose any of them. We are going to use $min = 0.5$.

9.3 Testing the Value of *rlFactor*

rlFactor	Graph 1	Graph 2	Graph3	Graph 4	Graph 5
2.8	347	1432	3379	595	62
2.9	302	1358	2682	498	60
3.0	283	1420	2094	408	73
3.1	289	1333	1597	348	57
3.2	321	1669	1454	330	62
3.3	312	1451	1400	264	56
3.4	303	1503	1242	236	69
3.5	307	1747	1112	232	61
3.6	357	1883	983	218	63
3.7	378	1763	877	209	68
3.8	425	1627	850	192	69
3.9	400	1755	742	206	77
4.0	483	2001	696	186	72
4.1	571	2387	677	182	74
4.2	587	2176	655	190	74
4.3	642	2304	605	187	78

In this case we can see some tendencies, thus connection between the parameter *rlFactor* and the produced results. 3.7 and 3.8 are the most optimal

parameters. We are going to use $rlFactor = 3.7$.

9.4 Testing of the Population and Mating Pool Sizes

There is no relation between the graph and the reproduction operations, so while testing the following parameters we use only graph 1.

Because of our reinsertion method (see 8.3) these parameters are continuous. We have to test whether elitist or generational genetic algorithms are better at solving our problem. Also we are going to find the optimal sizes and the optimal proportion of these parameters.

At first we are going to set population size to 50 and try few values for the mating pool size.

Mating pool size	40	45	48	49	50	55	60	70
95%	663	590	411	363	504	425	387	437

From these results we can see that elitist genetic algorithm is better for this problem. We are going to test some other population and mating pool sizes.

Population size is 40:

Mating pool size	35	38	39	40
95%	662	476	389	638

Population size is 100:

Mating pool size	95	96	97	98	99
95%	498	478	465	438	415

Some other result:

Population size / matingpool size	75 / 74	75 / 73	60 / 59	55 / 54	45 / 44
95%	380	385	367	387	368

It seems that a reasonable size for the population is 50 and for the mating pool is 49.

It seems that the most reasonable size of the population is 50 and for the mating pool it is 49.

After evaluating this test result we could think that elitist genetic algorithm with larger mating pool then the population size will lead to better results but after some further tests we can see that it won't. If the size of the population is 50 and we always put the best individual from the current generation to the next one we will get the following results:

Mating pool size	48	49	50	52	55	60
95%	403	363	406	377	373	365

If we use larger mating pool size we decrease the number of iterations needed to achieve the best results but we also perform more crossovers and mutations and we increase the lifetime of the population.

If we put the best two individuals from the current generation to the next generation we will get worse results.

9.5 The Number of Compared Individuals in Tournament Selection

At this section we are going to test the reasonable number of compared individuals in tournament selection.

k	Graph 1	Graph 2	Graph3	Graph 4	Graph 5
2	749	2253	999	380	107
3	337	1479	805	221	72
4	374	1417	1290	258	60
5	456	1668	2265	308	50
6	540	1388	N/A	413	48
7	629	1489	N/A	N/A	47
8	779	1836	N/A	N/A	48
9	N/A	N/A	N/A	N/A	46
10	N/A	N/A	N/A	N/A	50

In our case the fields with N/A are not interesting.

So at this case the most reasonable number of compared individuals in tournament selection is 3.

9.6 Number of Offspring Created by Crossover

At this section we are going to decide whether to create one or two offspring by crossover process. If we create one offspring by crossover from a pair of individuals then the 95% of tests take less time then 1352 ms. If we create two offspring this time is 359 ms

9.7 Maximal Number of Mutations

At this section we are going to test the reasonable number of maximal performed mutations in the genotype. If the maximal number of mutations is 1 then the 95% of tests will take less then 365 ms, if it is 2 then 1402 ms. For 3 it is 5283 ms. These indicate that the most reasonable number of performed mutations in a genotype is 1. It means we have to perform little changes to get the best results.

9.8 The Needed Time and Iterations

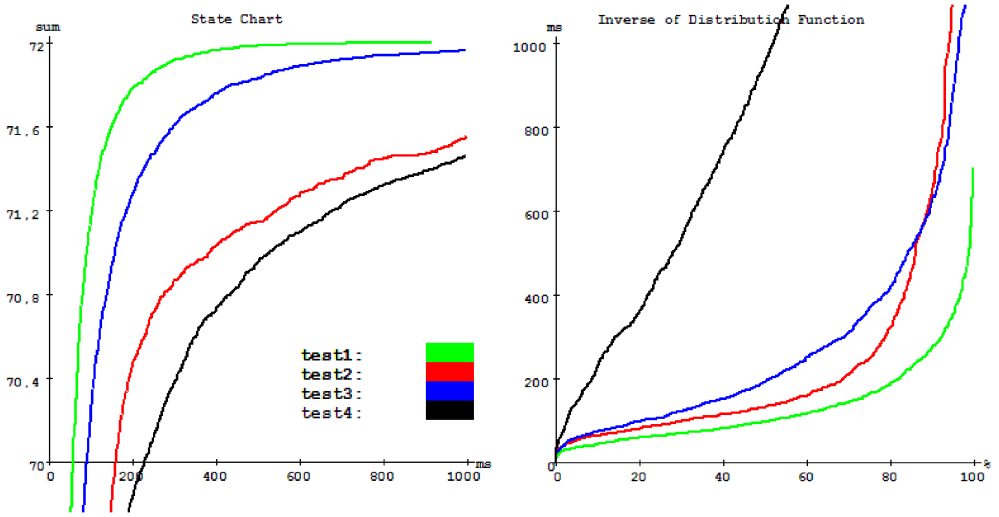
We will show the time and number of iterations needed to achieve the best result for each graph. We will use the above found values as parameters.

	50%	75%	90%	95%
Graph 1	99 ms / 64 iter	172 ms / 111 iter	254 ms / 164 iter	337 ms / 218 it
Graph 2	233 ms / 101 iter	455 ms / 197 iter	884 ms / 383 iter	1479 ms / 641 iter
Graph 3	254 ms / 140 iter	423 ms / 233 iter	659 ms / 363 iter	805 ms / 443 iter
Graph 4	89 ms / 42 iter	134 ms / 63 iter	189 ms / 89 iter	221 ms/ 104 iter
Graph 5	27 ms / 20 iter	42 ms / 32 iter	60 ms / 45 iter	72 ms / 54 iter

It means that if we don't have a very large graph with very large upper bound for the length of the graph 5000 iterations must be enough to find the best walk.

9.9 Some Charts

We will demonstrate the differences between the speed of optimization with some different parameters on Graph1:



At test1 we used the above defined values for the parameters.
At test2 we used $rlFactor = 2$.
At test3 we compared 2 individuals at tournament selection.
At test4 the maximal number of mutations was 3.

Chapter 10

Conclusion

The genetic algorithms are easy to implement but it is hard to find the exact algorithm and the correct parameters to get fast solution for all inputs. If we find a suitable algorithm it will be much faster than the traditional algorithms which is in our case is represented by backtracking.

This algorithm can be improved by finding the relationships between the input graphs and parameters like *rlFactor, c, max* and by developing a formula for computing the values of those parameters. Probably the perfect formula will be very complex if it even exists.

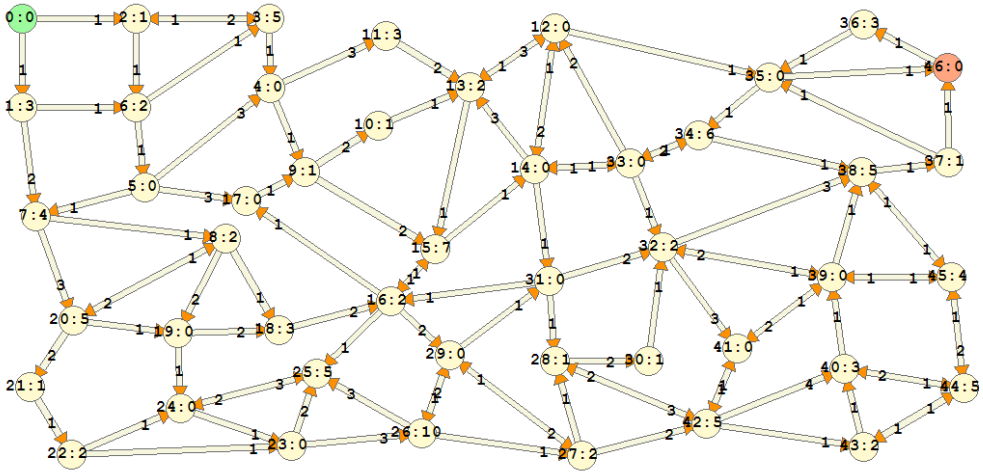
Chapter 11

Appendices

11.1 The Graphs

The following graphs were used at testes.

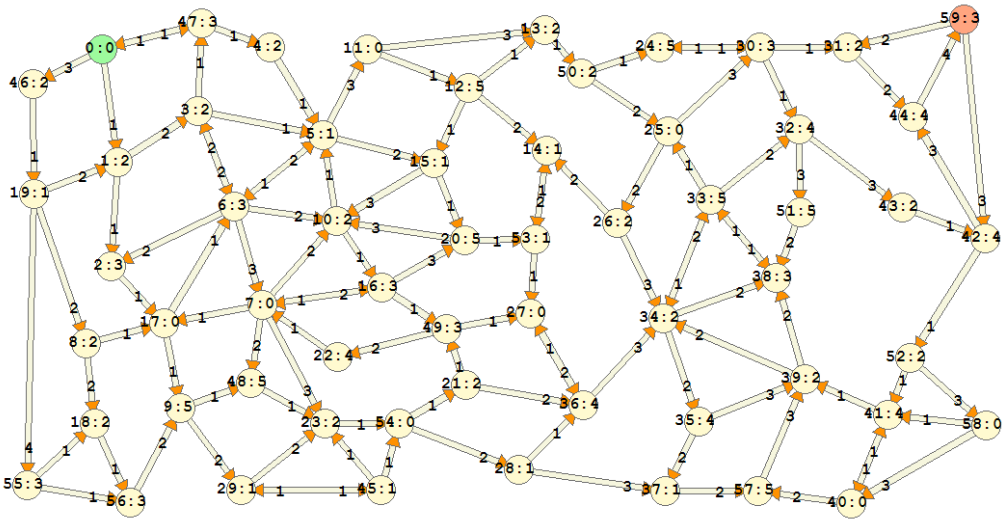
11.1.1 Graph 1



Used upper bound for the length of the walk is 30, the maximal sum of vertex weights is 72.

One of the possible maximal paths: 0, 1, 6, 3, 2, 6, 5, 7, 8, 18, 16, 15, 16, 29, 26, 27, 42, 43, 40, 44, 45, 38, 37, 35, 34, 38, 37, 46.

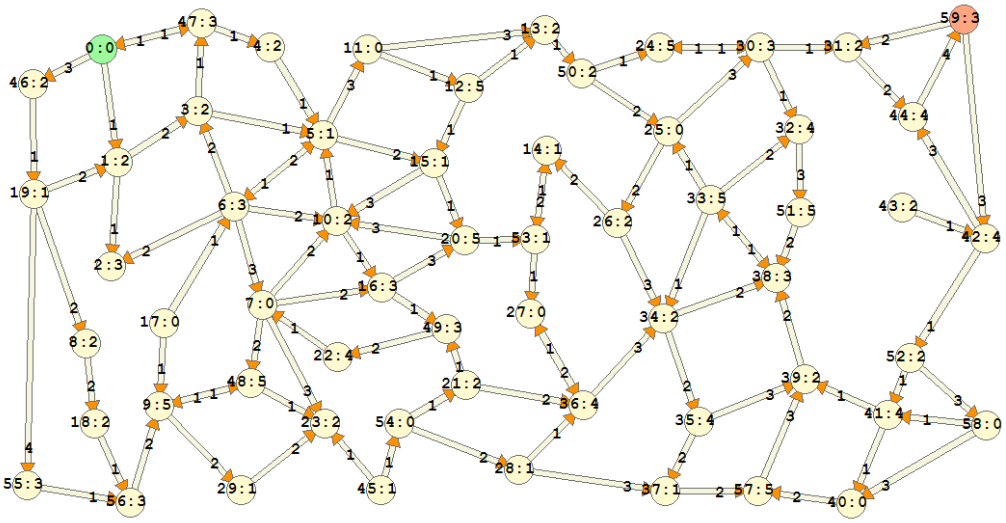
11.1.2 Graph 2



Used upper bound for the length of the walk is 80, the maximal sum of vertex weights is 120.

One of the possible maximal paths: 0, 1, 2, 17, 6, 3, 47, 0, 46, 19, 55, 18, 56, 9, 48, 23, 54, 21, 49, 22, 7, 16, 20, 10, 5, 11, 12, 13, 50, 24, 30, 32, 43, 42, 52, 41, 39, 34, 35, 37, 57, 39, 38, 33, 25, 30, 31, 44, 59.

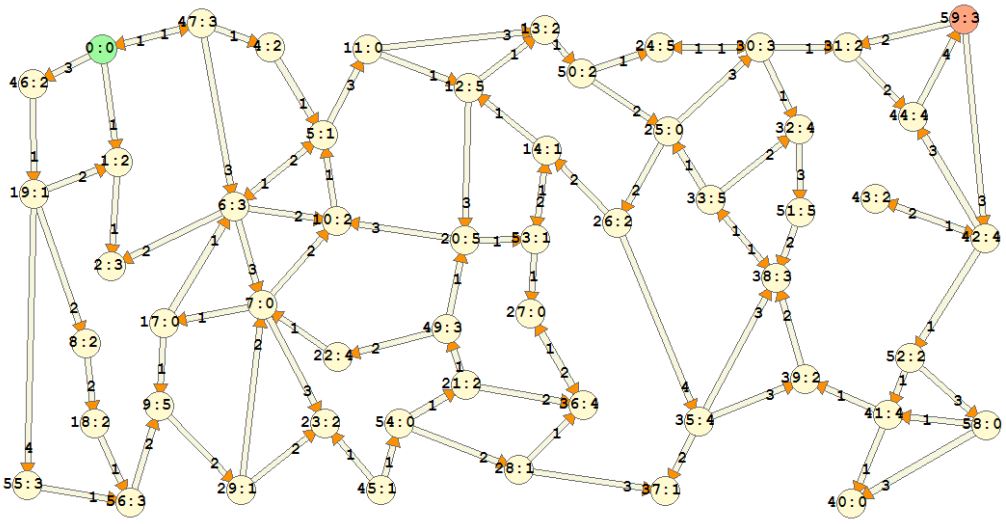
11.1.3 Graph 3



Used upper bound for the length of the walk is 50, the maximal sum of vertex weights is 74.

One of the possible maximal paths: 0, 1, 3, 47, 4, 5, 15, 20, 10, 5, 6, 10, 16, 49, 22, 7, 10, 5, 11, 12, 13, 50, 24, 30, 32, 51, 38, 33, 25, 30, 31, 44, 59.

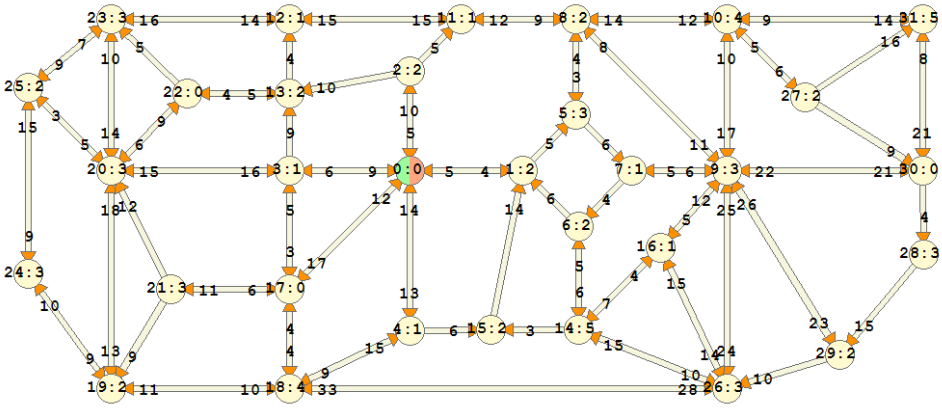
11.1.4 Graph 4



Used upper bound for the length of the walk is 60, the maximal sum of vertex weights is 77.

One of the possible maximal paths: 0, 47, 0, 46, 19, 8, 18, 56, 9, 29, 7, 17, 6, 10, 5, 11, 12, 20, 53, 14, 12, 13, 50, 24, 30, 32, 51, 38, 33, 25, 26, 14, 12, 13, 50, 24, 30, 31, 44, 59.

11.1.5 Graph 5



Used upper bound for the length of the walk is 130, the maximal sum of vertex weights is 36.

One of the possible maximal paths: 0, 3, 13, 22, 20, 25, 24, 19, 18, 17, 0, 1, 5, 8, 5, 7, 9, 16, 14, 6, 1, 0.

11.2 References

[1] A. Brindle (1981): “Genetic algorithms for function optimization.”

[2] Charles Darwin (1859): “On the Origin of Species”

[3] L. J. Fogel, A. J. Owens and M. J. Walsh (1966): “Artificial Intelligence through Simulated Evolution”

[4] John H. Holland (1975): “Adaptation in Natural and Artificial Systems”

[5] John R. Koza (1992): “Genetic Programming: On the Programming of Computers by Means of Natural Selection”

[6] John R. Koza (1994): “Genetic Programming II: Automatic Discovery of Reusable Programs”

[7] I. Rechenberg (1973): “Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologische Evolution”

[8] J. David Schaffer (1984): “Multiple Objective Optimization with Vector Evaluated Genetic Algorithms.”

[9] J. David Schaffer (1985): “Proceedings of the 1st International Conference on Genetic Algorithms and Their Applications”

[10] Stephen Frederick Smith (1980): “A Learning System based on Genetic Adaptive Algorithms.”

[11] Thomas Weise: “Global Optimization Algorithms – Theory and Application – (Version: 2009-06-26)

[12] A. Wetzel (1983): “Evaluation of the Effectiveness of Genetic Algorithms in Combinatorial Optimization.”